

James A. Storer
89 South Great Rd.
Lincoln, MA 01773
phone: 781-259-1198
email: storer@cs.brandeis.edu

December 11, 2004

ATTN: Patent Application 10/803,507

Mr. Jean B. Jeanglaude
United States Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450
phone: 571-272-1804

Dear Mr. Jeanglaude:

Regarding my patent application 10/803,507 (filed 3/18/2004), thank you for your letter and for talking with me on the phone today. Enclosed is my amended application; the corrections that have been made are:

1. References on pages 4 - 7 have been removed and put in form 1449 (4 pages, enclosed).
2. I have made the following corrections to the claims; these claims have "(amended)" after the claim number, all other claims have "(original)" after the claim number:

Claim 16: Delete " the steps of" (Line 2).

Claim 17: Delete " the step of:" (Line 18).

Claim 18: Replace "method" with "system" (Line 20).

Claim 19: Replace "method" with "system" (Line 21).

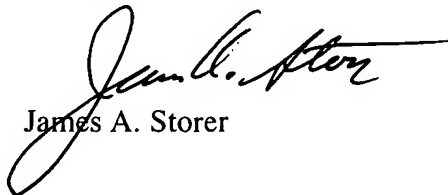
Claim 20: Replace "method" with "system" (Line 22).

Claim 22: Delete " the step of:" (Line 9).

Claim 23: Delete " an" (Line 13) and delete " method" (Line 14).

Thank you.

Sincerely,


James A. Storer

Enclosures (amended application and form 1449 pages)

RECEIVED
JAN 25 2005
TECHNOLOGY CENTER 2800

1 **Inventors:**

2 James A. Storer, Lincoln, MA

3 Dana Shapira, Brookline, MA

4 **Contact Address:**

5 James A. Storer

6 89 South Great Rd.

7 Lincoln, MA 01773

8 *phone and fax: 781-259-1199*

9 **TITLE OF THE INVENTION**

10 **In-Place Differential Compression**

11 **CROSS-REFERENCE TO RELATED APPLICATIONS**

12 *(Not Applicable)*

13 **STATEMENT REGARDING FEDERALLY SPONSORED**
14 **RESEARCH OR DEVELOPMENT**

15 *(Not Applicable)*

16 **REFERENCE TO A SEQUENCE LISTING, A TABLE, OR A**
17 **COMPUTER PROGRAM LISTING COMPACT DISK APPENDIX**

18 *(Not Applicable)*

BACKGROUND OF THE INVENTION

The present invention relates to in-place differential compression, where a body of data T is compressed with respect to a body of data S by performing copies from S in such a way that no additional memory is used beyond what is needed to store the longer of S or T ; that is, when decoding, S is overwritten from left to right as T is constructed in its place.

There have been many patents and technical papers that pertain to data compression. Many relate to techniques different than ones that employ string copying such as Huffman coding (e.g., U.S. patent 4,646,061) or arithmetic coding (e.g., U.S. patent 4,905,297). Many relate to techniques that employ string copies but in a traditional data compression model where a single body of data is compressed, not in-place differential compression of a first body of data with respect to a second body of data; for example, U. S. patents such as Holtz [4,366,551], Welch [4,558,302], Waterworth [4,701,745], MacCriskin [4,730,348], Miller and Wegman [4,814,746], Storer [4,876,541, 5,379,036], Fiala and Greene [4,906,991], George, Ivey, and Whiting [5,003,307, 5,016,009, 5,126,739], Rubow and Wachel [5,023,610], Clark [5,153,591, 5,253,325], Lantz [5,175,543], Ranganathan and Henriques [5,179,378], Cheng, Craft, Garibay, and Karnin [5,608,396], and technical articles such as Lempel and Ziv [1977, 1979] and Storer [1978, 1988, 1982, 2002].

There have also been a number of patents and technical papers relating to differential compression that do not perform decoding in-place; for example: Squibb [5,479,654, 5,745,906], Morris [5,813,017], Muralidhar and Chandan [6,233,589], Thompson, Peterson, and Mohammadioun [6,671,703], and technical articles such as Weiner [1973] (who developed a linear time and space greedy copy/insert algorithm using a suffix tree to search for matching substrings), Wagner and Fischer [1973] (who considered the string-to-string correction problem), Heckel [1978] (who presented a linear time algorithm for detecting block moves using longest common substring techniques), Tichy [1984] (who used edit-distance techniques for differencing and considered the string to string correction problem with block moves), Miller and Myers [1985] (who presented a comparison program for producing delta files), Fraser and Myers [1987] (who integrated version control into a line editor so that on every change a minimal delta is retained), Reichenberger [1991] (who presented a greedy algorithm for differencing), Apostolico, Browne, and Guerra [1992] and Rick [1995] (who considered methods for

1 computing longest common subsequences), Burns and Long [1997b] (use delta compression to
2 modify ADSM, Adstar Distributed Storage Manager of IBM, to transmit compact encodings of
3 versioned data, where the client maintains a store of reference files), Hunt, Tichy and Vo [1998]
4 (who combine Lempel-Ziv type compression and differential compression to compute a delta file
5 by using a reference file as part of the dictionary to compress a target file), Factor, Sheinwald
6 and Yassour [2001] (who present a Lempel Ziv based compression with an extended dictionary
7 with shared data), Shapira and Storer [2002] (who give theoretical evidence that determining the
8 optimal set of move operations is not computationally tractable, and present an approximation
9 algorithm for a block edit-distance problem), Agarwal, Amalapurapu, and Jain [2003] (who
10 speed up differential compression with hashing techniques and additional data structures such as
11 suffix arrays).

12 There has also been commonly available software available for differencing that does not
13 employ in-place decoding with string copying, such as the UNIX *diff*, *xdelta* and *zdelta* utilities.

14 Burns and Long [1997], M. Ajtai, R. Burns, R. Fagin, and D. D. E. Long [2002], and the U.S.
15 patent of Ajtai, Burns, Fagin, and Stockmeyer [6,374,250] use a hash table with Karp-Rabin
16 footprints to perform differential compression of one file with respect to another, using constant
17 space in addition to that used by both files, but do not provide for in-place decoding.

18 Burns and Long [1998], Burns, Stockmeyer, and Long [2002], and the U.S. Patent of Burns and
19 Long [6,018,747] present an in-place reconstruction of differential compressed data, but do not
20 perform the reconstruction with copies that overwrite from left to right. They begin with a
21 traditional delta file and work to detect and eliminate write-before-read conflicts (increasing the
22 size of the delta coding).

23 The invention disclosed here is in part motivated by the research presented in Shapira and Storer
24 [2003].

BRIEF SUMMARY OF THE INVENTION

Therefore, it is an object of this invention to perform *in-place differential compression*. With differential compression, a body of data T is compressed with respect to a body of data S . That is, an encoder and decoder have available identical copies of S , and the encoder may copy substrings from S to form T . A pointer that copies a string w from S achieves data compression when the bits representing that pointer are fewer than the bits representing w . We use the variable n to denote the size of T and m to denote the size of S . Differential compression is *in-place* if the memory containing S is overwritten when decoding T so that at no time is more than a total of $MAX\{m,n\}$ memory used, in addition to the constant amount of memory to store the program itself along with its local variables.

It is another object of this invention to rearrange the substrings of S to better align S and T to enhance the effectiveness of copying substrings from S to T in-place.

An in-place differential compression method according to the present invention includes the steps of rearranging substrings of S to improve the alignment of substrings of T with substrings of S and decoding in-place by copying substrings of S to form portions of T in a way that overwrites S .

1 **BRIEF DESCRIPTION OF SEVERAL VIEWS OF THE DRAWING**

2 FIG. 1 shows encoding on the top and on the bottom the way in which in-place decoding
3 overwrites memory.

4 FIG. 2 shows the well known method of sliding window compression (prior art).

5 FIG. 3 shows the of linking gaps when off-the-shelf compression follows aligned moves.

6

DETAILED DESCRIPTION OF THE INVENTION

With *differential compression*, a body of data S of size m is compressed with respect to a body of data T of size n . That is, both the encoder and the decoder have a copy of S and then a new string T may be encoded and subsequently decoded by making use of S (e.g. copying substrings of S).

There are many practical applications where new information is received or generated that is highly similar to information already present. When a software revision is released to licensed users, the distributor can require that a user must perform the upgrade on the licensed copy of the existing version. When incremental backups are performed for a computing system, differential compression can be used to compress a file with respect to its version in a previous backup, with respect to a similar file in a previous backup, or with respect to a file already processed in the current backup. Differential file compression can also be a powerful string similarity test in browsing and filtering applications.

One of ordinary skill in the art can understand that there are many ways that compressed data may be transmitted from an encoder to a decoder, including on a communications link, over a wireless connection, through a buffer, by transfer to and from a storage device, etc., and that the form of compressed data transmission from the encoder to decoder does not limit the invention herein disclosed.

Differential compression is *in-place* if the memory containing S is overwritten when decoding T so that at no time is more than a total of $MAX\{m,n\}$ memory used. Of course, the decoder has stored somewhere the executable code for the decoding program (possibly in read-only memory or hardware), which not is part of memory used when we refer to the computation being in-place; that is, in-place refers to the amount of memory used above and beyond the program code. The program code may also make use of some fixed number of local program variables (indices of for loops, etc.) which are also not part of the memory used when referring to the computation as being in-place. We allow the encoder to use more memory if needed. The restriction that the decoder must operate in-place is desirable because it reflects practical applications where the decoder may have unknown or limited resources.

1 It is an object of this invention to perform in-place differential compression with methods that
2 are both powerful (i.e., typically achieve high compression) and provide for fast and space
3 efficient decoding.

4 **MAX and MIN Notation**

5 We use the notation $\text{MIN}\{x,y\}$ to denote the smaller of x and y and $\text{MAX}\{x,y\}$ to denote the
6 larger of x and y .

7 **Big O Notation**

8 It is standard in the computer science literature to use *Big O notation* to specify how the amount
9 of time or memory used by a method increases as a function of the input size. For two functions f
10 and g , both of which map non-negative integers to non-negative integers, $f(x)$ is said to be
11 $O(g(x))$ if there exist two constants a and b such that for all integers $x \geq a$, $f(x) \leq b * g(x)$. For
12 example, if a parameter K is chosen in such a way that K is $O(\log_2(\text{MIN}\{m,n\}))$, then it must be
13 true that there exists two constants a and b which are independent of m and n (a and b remain the
14 same no matter what the values of m and n are) such that for all values of m and n for which
15 $\text{MIN}\{m,n\} \geq a$, $K \leq b * \log_2(\text{MIN}\{m,n\})$.

16 The notation $O(1)$ denotes a fixed constant. That is, $f(x)$ is $O(1)$ if there exists two constants a
17 and b such that for all integers $x \geq a$, $f(x) \leq b$; if we let c be the constant equal to the maximum
18 value of $f(x)$ in the range $0 \leq x \leq b$, then $f(x) \leq c$ for all integers $x \geq 0$. A big O constraint can be
19 combined with other constraints. For example, for a parameter K , saying that " $K < \text{MIN}\{m,n\}$
20 and K is $O(\sqrt{\text{MIN}\{m,n\}})$ " means that although K may be chosen to be a function of m and n (i.e.,
21 K is larger when $\text{MIN}\{m,n\}$ is larger), for all m and n $K < \text{MIN}\{m,n\}$, and *also*, there exists two
22 constants a and b such that $K \leq b * \sqrt{\text{MIN}\{m,n\}}$, for all m and n for which $\text{MIN}\{m,n\} \geq a$.

1 **Memory**

2 We use the term *memory* to refer to any type of computer storage, such as the computer's internal
3 RAM memory, a hard drive, a read-writable disc, a communications buffer, etc. We use the term
4 *character* to refer to the basic unit of data to which compression is being applied to a body of
5 data in some computer memory. A common example of a character is a byte (8 bits) or a set of
6 items that are stored one per byte (e.g. 7-bit ASCII codes). However, all of what we describe
7 applies as well to other types of characters such as audio, image, and video data (e.g., for audio
8 data that is stored using 12 bits per sample, the set of possible characters is the set of 4,096
9 distinct 12-bit values).

10 Although it is common for a body of data to be stored in sequential locations of memory, it is not
11 required for this invention, it could be that different portions of the data are stored in different
12 portions of memory. We assume that there is a linear ordering to a body of data (a first character,
13 a second character, a third character, etc.). If a character x occurs earlier than a character y in the
14 sequence of characters that comprise a body of data, then we say x is to the *left* of y , and y is to
15 the *right* of x .

16 The *size* of a body of data is the number of characters that comprise it, and corresponds to the
17 size of the memory used to store it (if the body of data resides in two or more disconnected
18 portions of memory, the size of the body of data is the sum of the size of each of the portions of
19 memory it occupies).

20 When we refer to the amount of memory used when decoding for in-place differential
21 compression of a body of data T with respect to a body of data S , it is always understood that we
22 are referring to the memory used to store portions of S and T , and that there may be an additional
23 fixed amount of memory used to contain decoding instructions and local variables used for
24 decoding (indices for loops, etc.).

25

1 **Sliding Window Compression**

2 Referring to FIG. 2, to help describe our invention, we first review the well known method in the
3 prior art of *sliding window compression*. The standard UNIX *gzip* utility is an example of a
4 sliding window compressor / decompressor. Given a string, sliding window compression
5 represents it with a smaller string by replacing substrings by a *pointer* consisting of a pair of
6 integers (d, l) where d is a *displacement* back in a window of the last N characters and l is the
7 *length* of an identical substring. The reduction in size achieved on a string depends on how often
8 substrings are repeated in the text, how the pairs (d, l) are coded, etc. Typically "greedy"
9 matching is used (always take the longest match possible).

10 There are many ways that have been proposed to encode pointers. A simple method based on
11 fixed length codes is:

- 12 • Displacements are represented with $\lceil \log_2(N) \rceil$ bits.
- 13 • Lengths range from 3 to some upper limit *MaxLength*, which can be represented by the
14 *MaxLength-2* integers 0 through *MaxLength-3*, and use $\lceil \log_2(MaxLength-2) \rceil$ bits.
- 15 • An initial flag bit distinguish a pointer (to a substring of 3 or more characters) from a
16 code for a single character, where the leading bit is a 0 if the next $\lceil \log_2(A) \rceil$ bits to follow
17 are a character and a 1 if the next $\lceil \log_2(N) \rceil + \lceil \log_2(MaxLength-2) \rceil$ bits to follow are a
18 displacement and length, where A denotes the size of the input alphabet.

19 For example, if $N=4,095$, *MaxLength*=10, and we use the 128 character ASCII alphabet
20 ($A=128$), a single character is represented with one byte (a flag bit and 7 ASCII bits) and a
21 pointer uses two bytes (a flag bit, 12 displacement bits, and 3 length bits).

22 Other methods may use variable length codes (some of which may be able to represent a string
23 of two bytes with less than 16 bits).

1 It is also possible to employ an off-the-shelf variable length coder to encode fields of a pointer.
2 For example, to compress data where very large matches may be present, one could use the
3 following coding method for pointers that employs an arithmetic coder:

- 4 • The pointer format begins with a control integer between 0 and 4 that indicates one of the
5 following cases:

6 *Case 0:* No displacement or length; the bits to follow indicate a single raw character.

7 *Case 1:* Displacements $< 4,096$ that point to a match of length < 256 .

8 *Case 2:* Displacements between 4KB and 36KB that point to a match of length < 256 .

9 *Case 3:* Displacements larger than 36KB that point to a match of length < 256 .

10 *Case 4:* Displacements larger than 36KB that point to a match of length ≥ 256 .

- 11 • Separate off-the-shelf arithmetic encoders are used to encode control integers, the raw
12 character for Case 0, and the length fields for Cases 1 through 3.

- 13 • A fixed code of either 12 or 15 bits is used for the displacements for Cases 1 and 2. A
14 fixed length $\lceil \log_2(N) \rceil$ code is used for the displacement fields of Cases 3 and 4 and for
15 the and length field of Case 4.

16

1 **The Present Invention**

2 A problem with previous methods for differential compression is their use of more
3 computational resources (e.g., time and memory) than may be necessary and / or the need to
4 sacrifice achievable compression in order to achieve acceptable resources. Here we propose a
5 high performance fast method for performing differential compression in place.

6 The solution proposed here uses the *In-Place Sliding Window* (IPSW) method to compress a
7 body of data T of size n with respect to a body of data S of size m :

8 *IPSW Encoding Algorithm:*

9 *Step 1:* Append T to the end of S .

10 *Step 2:* Compress T with a sliding window of size $\text{MAX}\{m,n\}$.

11 When the encoder slides a window of size $\text{MAX}\{m,n\}$, if m is equal to or larger than n , then only
12 at the start can pointers reach all the way back to the start of S , and as we move to the right, more
13 of S becomes inaccessible. If m is less than n , then after producing the first $n-m$ characters of T ,
14 it again becomes the case that pointers cannot reach all the way back to the start of S . In either
15 case, by overwriting the no longer accessible portion of S from left to right, decoding can be
16 done in-place, using $\text{MAX}\{m,n\}$ memory (encoding may use $m+n$ memory). That is, each time
17 an additional pointer is received by the decoder, it is decoded and the window slides right to add
18 characters to the right of T and remove them from the left of S . If $m < n$, then decoding increases
19 the size m of the memory to store S to a size n of memory to store T . If $m > n$, then after decoding
20 is complete, T resides in what used to be the rightmost n characters of S , and the memory for
21 what used to be the first $m-n$ characters of S can be reclaimed.

22 Of course, the decoding may use a fixed amount of memory for the decoding instructions (which
23 could be read-only memory or hardware) and a fixed amount of memory to contain a fixed
24 number of local variables used for decoding (indices of loops, etc.). This fixed amount of
25 memory is independent of m and n . Using big O notation, the memory used for decoding
26 instructions is $O(1)$ and the number of local variables is $O(1)$. When we say that decoding can be
27 done in-place using $\text{MAX}\{m,n\}$ memory, it is always understood that the memory used for
28 decoding instructions and local variables is in addition to this $\text{MAX}\{m,n\}$ memory.

Referring to FIG. 1, depicted is encoding (top) and decoding (bottom) for when the size of T is 50 percent larger than the size of S , and we are at the point when $n-m+x$ characters of T have been encoded. The hatched region is all of S on the top and the remaining portion of S on the bottom. The lightly shaded region is the $n-m$ characters of T that have already been encoded and decoded by the decoder without having to overwrite S . The dark shaded region is the portion of T that has already been encoded by overwriting the first x characters of S . The remaining $m-x$ characters of T have yet to be encoded, and hence the rightmost $m-x$ characters of S are still available to the decoder. The decoder's window can be viewed as two pieces: the already decompressed portion of T (the lightly shaded and darkly shaded regions) that is to the left of the pointer and the remaining portion of S that is to the right of the pointer (but was to the left of the lightly shaded and darkly shaded regions when encoding). So for each pointer decoded, at most two string copies must be performed: a single copy when the match is contained entirely in S (hatched region) or entirely in T (lightly shaded and darkly shaded regions) or two copies when the match starts in S and ends in T (the encoder encodes a match that crosses the boundary between the hatched region and the lightly shaded region). Since in many practical applications matches that cross this boundary are really just "lucky" (i.e., we may be primarily looking for large matches to S and if they are not found then a match in T that is a shorter distance away), an alternate embodiment of this invention is to forbid copies that cross this boundary, in order to further simplify decoding.

Any reasonable implementation of sliding window encoding can be used that allows for large matches. For example, the UNIX *gzip* utility that uses a window of size 32K and maximum match length 256; it is easy to modify the UNIX *gzip* utility to use a simple escape code for long pointers (e.g. reserve the longest possible length or displacement for the escape code) so that the same algorithm can be used for normal matches and the increased pointer length is only associated with large matches.

Another object of this invention is, when in some applications it may be that there is some additional amount of memory K available, to take advantage of this additional memory to improve the amount of compression achieved. It could be that K is a relatively small amount of memory that grows larger as m and n grow larger, such as K being $O(\sqrt{\text{MIN}\{m,n\}})$ or even K being $O(\log_2(\text{MIN}\{m,n\}))$, or it could be that K is relatively large, but still not large enough to

1 retain a separate copy of S while decoding (that is, $K < \text{MIN}\{m,n\}$), such as 50 percent of the
2 additional memory needed (that is, $K = \text{MIN}\{m,n\}/2$) or even 90% of the additional memory
3 needed (that is, $K = (9/10)\text{MIN}\{m,n\}$). An additional amount of memory K can be utilized to
4 potentially improve compression by lengthening the sliding window to size $\text{MAX}\{m,n\}+K$, thus
5 delaying by an additional K characters the point at which S begins to be overwritten when
6 decoding. That is, encoding begins with S as the rightmost characters of a window of size
7 $\text{MAX}(m,n)+K$.

8 In many practical applications, such as distribution of successive versions of a software
9 application, S and T are highly similar and reasonably well *aligned*; that is, large matches
10 between S and T occur in approximately the same relative order. In this case, IPSW can be a fast
11 method that performs in-place as well as methods that are not in-place.

12 Another object of this invention is to achieve good performance even when S and T are not well
13 aligned, by preceding IPSW with an additional step to improve alignment. Poor alignment could
14 happen, for example, with a software update where for some reason the compiler moved large
15 blocks of code around. An extreme case is when the first and second halves of S have been
16 exchanged to form T ($S = uv$ and $T = vu$); to decompress T the IPSW algorithm overwrites u as it
17 copies v , and then the ability to represent v with a single copy is lost. Rather than modify IPSW
18 (which is already a very fast and practical method that suffices for many and perhaps most
19 practical inputs), we propose a preprocessing stage for IPSW that moves substrings within S to
20 better align S with T . The compressed format can incorporate a way for the decoder to determine
21 whether preprocessing was performed (e.g., compressed data can be begin with an initial bit
22 indicating whether preprocessing has occurred). The encoder can compress T with IPSW and
23 compare that to compressing T with respect to S not in place (so at all times all substrings of S
24 are available to be copied). If the difference is significant, this initial bit can be set, alignment
25 preprocessing performed, and a list of moves prepended the normal IPSW encoding. The
26 decoder can perform the moves and then proceed in-place with normal IPSW decoding.

27 If the encoder determines that S and T are not well aligned, then the goal of preprocessing for the
28 IPSW algorithm is to find a minimal set of substring moves to convert S to a new string S' that is
29 well aligned with T . We limit our attention to moves that are non-overlapping, where the moves

1 define a *parsing* of S and T into *matched blocks*, $\{b_i\}_{i=1..r}$, and *junk blocks*, $\{x_i, y_i\}_{i=1..r}$; that is,
2 $S = x_0 \cdot b_{\alpha(1)} \cdot x_1 \cdot b_{\alpha(2)} \cdot x_2 \cdots x_{r-1} \cdot b_{\alpha(r)} \cdot x_r$ and $T = y_0 \cdot b_1 \cdot y_1 \cdot b_2 \cdot y_2 \cdots y_{r-1} \cdot b_r \cdot y_r$. When using the sliding window
3 method, we would like to copy substrings of S only from the part that was not yet overwritten by
4 the reconstruction of T . That is, we would like to perform only *left copies*, i.e., a copy (s_i, d_i, l_i)
5 that copies a substring with l_i characters from position s_i to position d_i that satisfies $s_i \geq d_i$.

6 We can improve upon the idea of greedy block-move edit distance computation proposed in
7 Shapira and Storer [2002] by using two different kinds of rearrangements of the blocks. *Moves*
8 rearrange the blocks $\{b_i\}_{i=1..r}$ so that they occur in S and T at the same order. *Jiggings* move the
9 junk blocks of S backwards, so that, as a side affect, the matched blocks are moved forwards.

10 *Move Preprocessing Algorithm:*

11 Step 1: Find Non Overlapping Common Substrings (NOCS) of S and T from longest to
12 shortest down to a minimum length *stoplevelength*. These are the $\{b_i\}_{i=1..r}$ blocks.

13 Step 2: Compute the minimum number of rearrangements (moves and jiggings) in S so
14 that the blocks $\{b_i\}_{i=1..r}$ are left copies within S and T .

15 Step 3: Generate S' according to step 2.

16 Step 4: Compute $\text{IPSW}(S', T)$.

17 Step 1 can be performed by repeatedly applying a longest common substring computation (see,
18 for example, the text book *An Introduction to Data Structures and Algorithms*, by J. A. Storer,
19 Birkhauser / Springer, 2002), or by using the linear time method of M. Meyerovich, D. Shapira,
20 and J. A. Storer ("Finding Non-Overlapping Common Substrings in Linear Time", Technical
21 Report CS-03-236, Comp. Science Dept., Brandeis University); *minlength* can be tuned for
22 different applications, where a value of 256 has been found to be a good general practical choice.
23 Since the encoder is not restricted to work in-place, generating S' in Step 3 can be done in linear
24 time by copying the strings to a different location. Step 4 uses the fast linear time IPSW
25 algorithm that has already been described as one of the objects of this invention.

26 We now describe how Step 2 can be done in quadratic time of the number of blocks, which is in
27 the worst case $O((n/\text{stoplevelength})^2)$; in practice, the number of blocks is much smaller than \sqrt{n} , and
28 Step 2 works in linear time. The NOCS found in Step 1 are renamed using different characters,

1 and then the method of Shapira and Sorer [2002] can be performed to compute minimum edit
 2 distance with moves to attain the moved characters for the minimum cost which correspond to
 3 moving the NOCS. Our next goal is to produce a (source, destination) format for the NOCS
 4 moves to be sent to the decoder based on the character moves. For example, when dealing with 5
 5 NOCS renamed by $\{1,2,3,4,5\}$ one possibility to obtain 12345 from 15432 by character moves is
 6 by moving 2, 3, and 4, where 4 is moved backwards. Another option for transforming 15432 to
 7 12345 is by moving 3, 4 and 5, where 4 is moved forwards. Each one of these solutions can be
 8 obtained from the dynamic-programming table from different optimal paths going from cell $[r,r]$
 9 (for integers 1 through r) back to cell $[0,0]$, from which their alignment can be extracted. The
 10 source and destination positions depend on the order the moves are performed. Therefore, the
 11 encoder performs and reports one character move at a time, updating the source and destination
 12 locations, before proceeding with the next move. We define each item's destination to be the
 13 corresponding positions just after the last aligned item to its left. If there is no such item, the item
 14 is moved to the beginning of the array. The move causes a shift of all items to its right, and the
 15 item is marked as an aligned item after the move is performed. Let $\{b_i\}_{i=1..r}$ be a set of r matched
 16 blocks to be moved from source positions s_i to destination position d_i , and let $\{x_i\}_{i=1..r}$ be the
 17 'junk' blocks we wish to jiggle to the beginning of S , so that all blocks $\{b_i\}_{i=1..r}$ perform only left
 18 copies, i.e., $s_i \geq d_i$. After performing the block moves in S to obtain S' , we have $S' =$
 19 $x_0 b_1 x_1 b_2 x_2 \dots x_{r-1} b_r x_r$, and $T = y_0 b_1 y_1 b_2 y_2 \dots y_{r-1} b_r y_r$. To see that it is always possible to perform
 20 jiggings so that the matched blocks become left copies consider the worst situation, where all
 21 blocks $\{b_i\}_{i=1..r}$ are shifted all the way to the right, without ruining their order obtained from the
 22 edit-distance algorithm. Since the space available is at least n , we are left with only left copies.
 23 Thus, in the worst situation we perform $r-1$ moves and r jiggings. We are interested in
 24 minimizing this number. Each block that was moved already in the edit-distance with move
 25 algorithm, is farther moved to the right, so that it is now adjacent to its right non-junk neighbor.
 26 These moves are done for free, since they can be moved to the final point directly. At each stage
 27 of the jiggling algorithm, when we reach an illegal copy (i.e., a right copy), we choose the
 28 longest junk block to its right, and move it to the left of the leftmost illegal copy, which results in
 29 shifting all other blocks to the right.

1 A further object of this invention is for the decoder to perform the move pre-processing in-place.
2 Let S be a string of length m , and x, y denote the source and destination locations of a substring s
3 of S of length l to be moved. Since the problem is symmetric for left versus right moves, and
4 since we can easily incorporate an offset, assume $x=0, y>0$, and y specifies the character that
5 follows s after the move (e.g., if $S = abcdefgxyz$, $l=3$, and $y=7$, then abc is moved so that $S =$
6 $defgabcxyz$). A naive algorithm, that uses $O(1)$ additional memory and $O(y^2)$ time, moves the
7 characters of s individually, shifting the entire string between the source and destination
8 locations at each step.

9 For a more efficient algorithm, observe that each character of s goes a distance $d = y-l$, and the
10 move is tantamount to rearranging positions 0 through $y-1$ by the permutation $i \rightarrow (i+d) \text{ MOD } y$.

11 A standard in-place permutation algorithm (e.g., see the book of Storer [2002]) starts with
12 positions 0 through $y-1$ "unmarked" and then for each position, if it is unmarked, follows its
13 cycle in the permutation and marks each position visited, for a total of $O(y)$ time and y additional
14 bits of memory for the mark bits. Here, mark bits for only $\text{MIN}\{l, d\}$ positions are needed, since
15 every cycle passes through at least one of the positions 0 through $l-1$ and at least one of the
16 positions l through $y-1$. If l or d is $O(\log(m))$, then we use $O(1)$ memory under our model, since
17 we are assuming at least enough memory to store $O(1)$ local variables, which are each capable of
18 holding $O(\log(m))$ bits (e.g., for $m < 4$ billion, 32 bits suffice to store an index variable in a loop
19 that counts from 1 to m). If an additional amount of memory K is available that suffices to store
20 these bits, then we can employ this additional memory for this purpose. Otherwise, in
21 $O(y^{1/2} \log(y))$ time we can test $y, y-1, y-2, \dots$ until we find the largest prime $p \leq y$; that is, $d^{1/2}$
22 operations suffice to test, and approximately $\ln(y)$ positions are tested, where \ln denotes the
23 natural logarithm (this follows from the classic the "Prime Number Theorem" which appears in
24 many basic texts on number theory). Then, let $d' = p-l$, and using no mark bits, move s most of
25 the way by the permutation $i \rightarrow (i+d') \text{ MOD } p$ (since p prime implies that the permutation is a
26 single cycle that we can traverse until we return to the start). Finally, after adjusting the offset so
27 that s starts at position 0 , move s to position $y' = l+(y-p)$ with the permutation $i \rightarrow (i+(y-p))$
28 $\text{MOD } y'$; since $(y-p) \approx \ln(y) = O(\log(m))$, again, this can be done with $O(1)$ additional
29 memory in our model (i.e., we are assuming at least enough memory to store $O(1)$ local
30 variables, which are each capable of holding $O(\log(m))$ bits).

1 Another object of our invention is to provide the possibility of using an existing "off-the-shelf"
2 compression method after a set of strongly aligned moves has been performed on S . We say that
3 a set of substring moves performed on S to create a string $S2$ is strongly aligned if when we write
4 $S2$ below S and draw a straight line from each character in S that is moved to where it is in $S2$,
5 then no lines cross.

6 We can perform all strongly aligned moves for a string of length m in a total of $O(m)$ time and
7 $O(1)$ additional memory, using only simple string copies (that may overwrite characters that are
8 no longer needed). We scan twice, first from left to right to copy blocks that go to the left and
9 then from right to left to copy blocks that go to the right. Let x_i , y_i and l_i , $1 \leq i \leq n$, denote the source
10 and destination locations and the lengths of k aligned move operations for a string stored in the
11 array A :

```
12   for  $i:=1$  to  $n$  do if  $i$  is a source position such that  $x_i > y_i$  then  
13       for  $j:=1$  to  $l_i$  do  $A[y_i + j] := A[x_i + j]$   
14   for  $i:=n$  downto  $1$  do if  $i$  is a source position such that  $x_i < y_i$  then  
15       for  $j:=l_i$  downto  $1$  do  $A[y_i + j] := A[x_i + j]$ 
```

16 Any reasonable encoder (which need not work in-place) to construct a sequence of aligned
17 moves may suffice in practice for applications where large aligned moves exist. For example, a
18 greedy approach can parse the text twice, using two thresholds: C_1 and C_2 , where C_1 is much
19 larger than C_2 . First extract matches longer than C_1 , and then search between two subsequent
20 aligned matches for matches longer than C_2 .

21 Assuming that a move operation is represented by three components (source position, destination
22 position, and length), after the preprocessing step is complete, and the decoder has performed all
23 of the move operations, the gaps can now be filled in by running a standard decompressor on the
24 remaining bits so long as we are careful during preprocessing to remember these positions. This
25 can be done by linking them together as depicted in FIG. 3; the encoder can be implemented to
26 insure that each gap is large enough to hold a pointer (e.g., 4 bytes to handle moves of 4 billion
27 characters). From the point of view of the off-the-shelf decoder, a contiguous string is produced,
28 which we just happen to partition to fill the gaps.

1 Another way to use the off-the-shelf compressor is to modify it slightly so that gaps are
2 compressed in their local context. For example, with a sliding window method, the window
3 could be defined to go contiguously back to the left end of the current gap and then continue on
4 back into the previous block.

5 A third way to use the off-the-shelf compressor is to use a single pointer value as an escape
6 (followed by a displacement and length) to a match into the decoded string rather than into some
7 additional memory used by the off-the-shelf encoder / decoder that is in addition to the
8 $\text{MAX}\{m,n\}$ memory normally allocated for in-place decoding; for example, this would be
9 another way to make use of an additional amount K of memory.

10 Although this invention has been primarily described as a method, a person of ordinary skill in
11 the art can understand that an apparatus, such as a computer, could be designed or programmed
12 to implement the method of this invention. Similarly, a person of ordinary skill in the art can
13 understand that there are many types of memory that can be used to implement memory for the
14 purpose of this invention (RAM memory, hard drive, read-write disk, data buffers, hardware
15 gates, etc.).

16 Since certain changes may be made in the above methods and apparatus without departing from
17 the scope of the invention herein involved, it is intended that all matter contained in the above
18 description or shown in the accompanying drawings shall be interpreted in an illustrative and not
19 in a limiting sense.

20

1

SEQUENCE LISTING

2

(Not applicable.)

1 OATH OR DECLARATION

2 *(See the attached form,*

3 *"Declaration for Utility or Design Patent Application, 37 CFR 1.63.")*

4